macros

- Macros in a general: code elements allowing us to define expansions to an existing language
- Typically evaluated as a pre-processing step by the compiler/interpretter evaluating HLL source code
- Different languages provide very different macro capabilities
  - Some simply use macros to create aliases/shortcuts for other commands
  - Some allow simple rewrites/substitution of command syntax (like C #defines) or template-based code specification (like C++ templates)
  - Some provide full programmatic code rewrites (e.g. lisp)

Lisp macros

- Lisp macros look somewhat like functions but run in a preprocessing step, before execution of the lisp code
- Allow programmer to specify new syntax to be used in the source code, and detail how to translate that new syntax into regular lisp code
- During processing, where the compiler sees the new syntax in use it runs the macro to translate the new syntax
- After all macros have completed, the rewritten lisp code can be run

Simple example: nullify

- I want to be able to use syntax (nullify x) in my code to set a variable’s value to nil, e.g.. if I write (nullify foo) then during pre-processing it should get translated into (setf foo nil)
- I define a macro for nullify, specifying it takes one parameter and specifying what the rewritten code should look like, e.g.

```
(defmacro nullify (varname) `(setf ,varname nil))
```

- We’ll discuss the relevance of ` and , shortly

Nullify continued

- Suppose my source code looks like this

```
(defmacro nullify (varname) `(setf ,varname nil))
(defvar foo 3)
(nullify foo)
```

- Then after preprocessing, just before execution, it has been transformed into this

```
(defvar foo 3)
(setf foo nil)
```

Why not just call a function

- Why not just use a function called nullify? e.g.

```
(defun nullify (varname) (setf varname nil))
(nullify foo)
```

- Two key reasons:
  - a) pass by value, calling (nullify foo) wouldn’t change foo
  - b) the source code produced by the macro doesn’t need a function call during execution, so is more efficient at run time

So why use functions at all?

- Having a macro rewrite our source code before execution introduces an extra layer of indirection between the developer and the code that actually runs
- The rewritten code is what gets executed, so all the lisp error messages are based on what the code looks like after the macros finish ... not what the source code looks like to the developer
- This can make debugging very tricky, especially since macros can use other macros and can even be recursive

When should I use macros?

- When the use of a macro can significantly improve the readability of your code by providing a simple syntax for something the developer does frequently
- When you need to improve run time performance (avoiding runtime function calls)
- When you need to accomplish something that cannot be easily accomplished through function calls

So why the comma and backtic?

- Macros process our source code, translate it into other code
- For the macro to use variables/parameters, it needs a way to tell the preprocessor when to use literal text and when to embed a variable's value: when does it mean the word foo vs when does it mean use the value stored in foo?
- The backtic ` means whatever comes next is the literal text to generate, e.g. `` `(blah blah blah) `` would actually generate source code `(blah blah blah)`
- The comma is used inside a backtic to say "use the actual variable value here", e.g. ,x means use the value of x

Nullify revisited

- Our nullify example was

```
(defmacro nullify (varname) `(setf ,varname))
(nullify foo)
```

- Here the symbol foo is the value stored in the macro parameter varname, and gets used in producing the rewritten source code
- we can see the macro uses `(setf ....)` literally but substitutes the value in varname (i.e. foo), producing

```
(setf foo)
```

Fancier nullify

- Our macros can generate code that is as simple or as complex as desired
- The generated code could be a statement or block that includes error checking, e.g. check the var exists:

```
(defmacro (varname)
   `(if (boundp ,varname) (setf ,varname nil)))
(nullify foo)
```

- Would rewrite as

```
(if (boundp foo) (setf foo nil))
```

Seeing our expanded macro

- Sometimes in debugging it helps to see what code a macro call is generating, macroexpand-1 shows us this

```
(macroexpand-1 ‘(nullify foo))
```

- Would return the list

```
(if (boundp foo) (setf foo nil))
```

- Unfortunately, if used on recursive macros it only shows the first layer of expansion

Local variables and gensym

- Suppose I try the following swap macro using a local variable tmp

```
(defmacro swap (a b)
      (let ((tmp ,a)) (setf ,a ,b) (setf ,b tmp)))
```

- Then `(swap x y)` would translate into

```
(let ((tmp x)) (setf x y) (setf y tmp))
```

- That looks good, but what if I already had a tmp variable and tried `(swap tmp z)`, which would translate to

```
(let ((tmp tmp)) (setf tmp z) (setf tmp tmp))
```

gensym: unique identifiers

- No matter what name we pick for the macro’s local variable, there is a chance it will clash with an existing variable name in the user program
- (gensym) is a function that returns a symbol guaranteed not to clash with anything else in the program
- We’ll have the macro call gensym, store the symbol it gives back in a local variable, then use that variable as part of our code rewrite

Swap using gensym

- We introduce a let block in the macro to store the name gensym creates for us, then embed the stored name into the generated code:

```
(defmacro swap (a b)
   (let ((tmp (gensym)))
      `(let ((,tmp ,a)) (setf ,a ,b) (setf ,b ,tmp))))
```

- for `(swap x y)` suppose the unique symbol name created by gensym was (for example) G123, the result would be:

```
(let ((G123 x)) (setf x y) (setf y G123))
```

Embed content of a list with ,@

- Suppose I wanted a macro that could dump the contents of a list into an expression
- e.g. (really simple example) I want a “plus” macro so that

  `(plus ‘(10 20 30))` gets rewritten as `(+ 10 20 30)`
- If I try `(defmacro plus (L) `(+ ,L))`
- It would still just produce `(+ ‘(10 20 30))`
- We can use ,@ instead of , to produce the desired effect, e.g.

```
(defmacro plus (L) `(+ ,@L))
```

Recursive macros with &rest

- We can use &rest in the macro parameter list to specify a variable number of parameters
- The macros can also be made recursive
- Example: multi-parameter AND implemented using nested if statements:
  - Translate (and w) into w
  - Translate (and w x) into (if (w) x)
  - Translate (and w x y) into (if (w) (if (x) y))
  - Translate (and w x y z) into (if (w) (if (x) (if (y) z)))

Implementation of and

```
(defmacro AND (&rest args)
   (cond
      ; translate (AND) to just t
      ((null args) `t)

      ; translate (AND x) to just x
      ((null (cdr args)) ,(car args))

      ; general case (AND x ...more...) translates to
      ;    (if (x) ...result of recursive macro call...)
      (t `(if (,(car args)) (AND ,@(cdr args))))))
```

Most lisp syntax actually macros

- Most of the lisp features we use are actually macros that translate our code into core lisp elements
- This includes functions like and, or, the various forms of loops, cases, cond, blocks and let blocks, etc etc etc
- By incrementally adding macros we expand the syntax available to the developer without increasing the complexity of the underlying language
- You can create your own flavour of lisp syntax by building up your own macro suite